Scheduling in the REAKT kernel: combining predictable and unbounded computations for maximising solution quality in real-time knowledge-based systems

A Mensch, François Charpillet

Scheduling in the REAKT kernel : combining predictable and unbounded computations for maximising solution quality in real-time knowledge-based systems

A. Mensch and F. Charpillet***
**Thomson-CSF RCC, 160 Bd de Valmy, F-92704 Colombes Cedex*
***CRIN-INRIA, PO Box 239, F-54506 Vandoeuvre-les-Nancy*

Résumé : La complexité croissante des applications temps réel a renouvelé l'intérêt porté aux techniques de l'Intelligence Artificielle dans ce domaine. Cet article présente le Noyau d'Exécution REAKT, un environnement logiciel adapté aux besoins des applications temps réel à base de connaissances, et en particulier les techniques d'ordonnancement utilisées pour faciliter l'exécution conjointe de traitements déterministes et non déterministes tout en garantissant les temps de réponse.

Les techniques utilisées reposent sur le principe de raisonnement progressif, une approche qui permet à une application d'obtenir rapidement une première solution, puis de l'affiner tant que le temps disponible le permet. Un modèle de tâches est proposé pour permettre l'intégration aisée d'actions reflexes et de raisonnements complexes. Un algorithme permettant de maximiser le temps disponible pour les traitements complexes non déterministes tout en garantissant les temps de réponse des tâches temps réel. Cet algorithme fournit une réponse au problème d'ordonnancement joint de tâches périodiques, sporadiques et optionnelles.

Mots-clés : Systèmes temps réel à base de connaissances, ordonnancement.

1. THE NEED FOR MORE " INTELLIGENT " REAL-TIME SYSTEMS

Over the last few years, the usefulness and maturity of Artificial Intelligence technologies, and in particular of knowledge-based systems, have been demonstrated by the ever growing number of real-life applications using them. In the domain of real-time systems (process control, communication network management, satellites and avionics, patient monitoring, C^3I systems, robotics, data analysis...), these techniques have been considered with a renewed interest, as they appear as a promising approach to cope with the increasing complexity of the systems to be controlled [Laffey et al. 88].

Although covering a wide range of domains, complex real-time applications show several common features :

- They are tightly coupled with a physical system (plant, network, satellite, robot, human body, battlefield...) which has to be controlled. This system evolves dynamically under the influence of external constraints which are not completely known by the control application. The behaviour of such an application is therefore strongly directed by the evolution of the controlled system, which generates data throughput and response times requirements.
- They run in an environment which presents characteristics difficult to handle for classical real-time systems. This environment is often very dynamic, as in the case of communication networks or industrial processes, ill-modelled, as in the case of patient monitoring or financial analysis systems, or ill-defined, as in the case of autonomous vehicles or satellites. Such an environment increases the complexity of the control applications, as it requires from the application the ability to adapt itself to the evolving environment and to handle correctly time-varying, fuzzy, uncertain and incomplete information.
- They have to provide results which go beyond the capabilities of today's real-time systems. Although current applications perform low-level tasks (such as data acquisition, data filtering and processing, detection of abnormal behaviour) very well, they usually depend on a human operator for higher-level tasks (such as diagnosis, situation assessment, action planning), as those tasks require a deep understanding of the mechanisms involved in the evolution of the controlled system. This approach becomes increasingly difficult with the growing complexity of the controlled systems, for practical, safety and economical reasons. These complex

applications thus require the integration of advanced technologies in classical real-time systems in order to better assist, and sometimes even replace, the human operator in the decision process.

We present in this paper some results of the REAKT and REAKT II ESPRIT projects (EP 5146 and 7805). The primary objective of those projects was to develop a set of tools and the associated methodology for applying knowledge-based systems in real-time domains. The projects were to produce definitions, specifications and prototypes of various techniques, to be eventually integrated into a toolkit for the efficient development, deployment and maintenance of knowledge-based modules that can be embedded into real-time applications.

Section 2 presents an overview of the REAKT Real-Time Kernel, a software architecture well-adapted to the execution requirements of real-time knowledge-based systems. Section 3 describes the model of application activities which is used for scheduling tasks in the kernel. Sections 4 to 7 then detail the algorithms which support the joint scheduling of predictable real-time tasks and unbounded optional computations. Finally, Section 8 presents some conclusions on advantages of the approach.

2. THE REAKT KERNEL

REAKT is a software environment for developing and delivering real-time knowledge-based systems (RTKBSs). The REAKT environment provides both tools and a complete methodology to ensure the efficient development of applications with the requirements presented in the previous section. This environment is made of three main components:

- REAKT gets its unique real-time capabilities from its *Real-Time Kernel*, which provides all the necessary techniques for co-operative and real-time problem solving.
- The REAKT *Development Toolbox* allows the REAKT user to populate the generic kernel with application-specific classes, objects and knowledge bases.
- The REAKT *Application Methodology* consists of a set of guidelines and support tools to assist the REAKT user throughout the application development life cycle.

This paper focuses on the REAKT Real-Time Kernel. More information on the REAKT Development Toolbox and the REAKT Application Methodology can be found respectively in [Galan et al. 93] and [Fjellheim et al. 94].

The REAKT Kernel [Mensch et al. 94] is based on the *Blackboard model* [Engelmore and Morgan 88] [Jagannathan et al. 89] of multi-agent co-operation. Blackboard systems offer a powerful and flexible problem-solving approach, with outstanding integration and control capabilities. A Blackboard system is a set of knowledge sources collaborating with each other through a shared memory area, called the Blackboard, in order to reach a solution to a problem. A separate control module directs the problem-solving process by deciding which knowledge source is most appropriate for execution, given the state of the solution.

The Blackboard approach is widely recognised as a powerful framework which allows:

- the scaling limitations of expert systems to be overcome through a modular approach;
- heterogeneous knowledge representation formalisms to be integrated;
- the application development life cycle and runtime execution to be better controlled.

While providing the benefits of the Blackboard approach, REAKT extends the classical model along several directions to provide an architecture adapted to real-time operation.

A REAKT application is composed of a set of tasks (Figure 2.1), each task being an *independent thread of control* in charge of executing one or several user-defined knowledge sources to contribute to the global solution of a problem. One or several input tasks can also be defined to perform data acquisition. Tasks can be either permanent or created dynamically in response to events. Tasks can be associated with *deadlines*, which represent the time at which the solution provided by a task must be available to avoid a system failure.

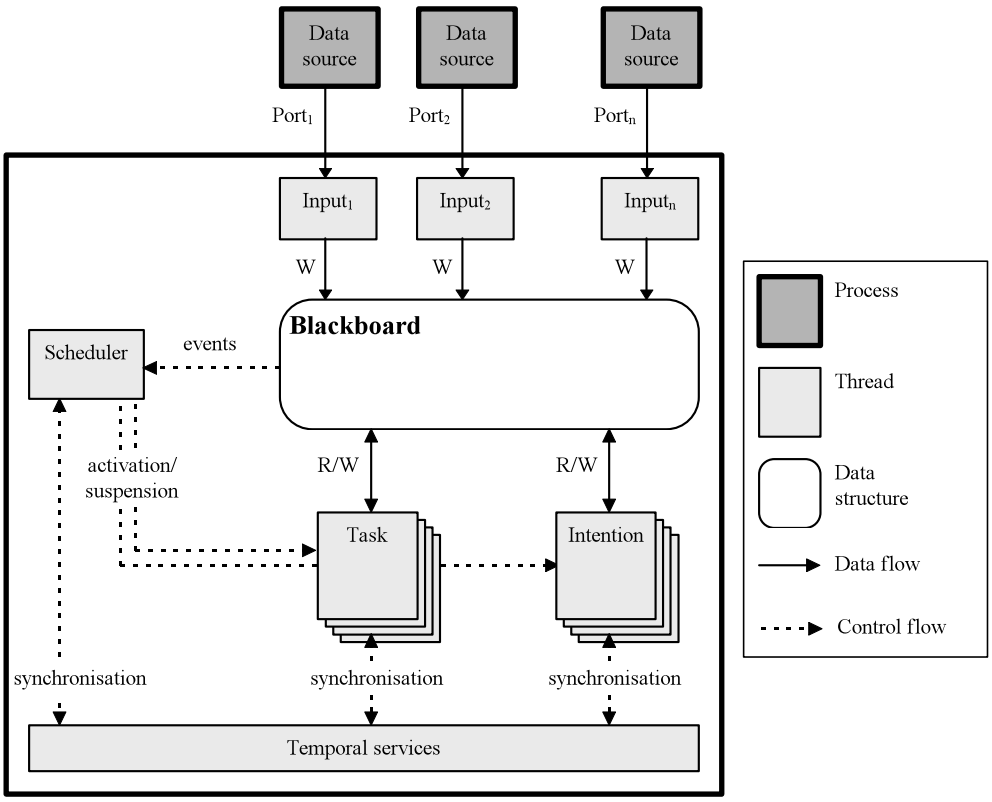

Figure 2.1 - Architecture of the REAKT Real-Time Kernel

REAKT uses a control strategy for task scheduling based on deliberative scheduling techniques [Garvey and Lesser 94] that offer efficient mechanisms to trade off computation time for quality of result. The proposed model follows the *progressive reasoning* paradigm [Charpillet and Boyer 94]. Progressive reasoning allows a system to obtain a first solution quickly, and then refine it while time is available. Two main types of tasks can be defined in a REAKT application:

- Real-time tasks, which must be permanent tasks with a bounded worst-case execution time and a task activation pattern which is either periodic or features minimal interarrival times. With these hypotheses, the REAKT scheduler is able to determine off-line whether a schedule meeting all real-time task deadlines is feasible, thus *guaranteeing all hard deadlines in an application*. Typical activities which can be performed at the first level include data acquisition, recognition of abnormal situations, reflex actions, output,...
- Optional tasks (called *intentions* in Figure 2.1), which improve the solution quality while time is available, using the opportunistic control mechanism of Blackboard systems. Optional tasks are either refinements of real-time activities, or dynamically created tasks with only soft deadlines. These tasks are scheduled according to their

deadlines and the quality of the solution already obtained, but their execution cannot be guaranteed, as neither their number nor their execution time need to be bounded.

3. THE REAKT TASK MODEL

REAKT provides a task model to describe the activities of an application. The introduction of a task model is required for two main reasons:

- No guarantee on the response times can be provided if the worst-case event arrival pattern in the application cannot be precisely estimated, i.e. the maximum load of the system is unknown.
- The progressive reasoning approach requires related real-time and optional tasks to be declared to the system.

In the REAKT task model, each task τ is decomposed into three parts: *mandatory*, *optional* and *action* (Figure 3.1) [Audsley et al. 91]. The mandatory part, activated at Rm_τ, is in charge of providing a first-level solution with a guaranteed response time; the optional part then tries to improve this result, using more complex reasoning mechanisms; the action part is activated before the task deadline D_τ, possibly pre-empting the optional part, and uses the best available solution to execute the necessary actions.

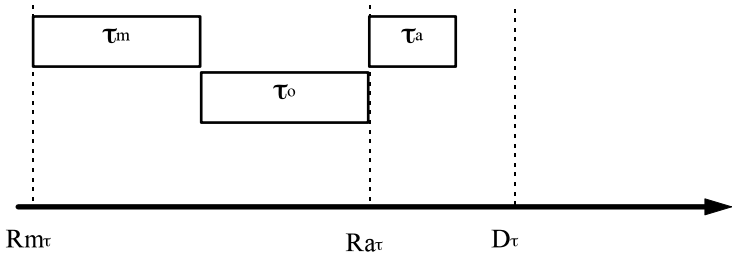

Figure 3.1 - Decomposition of a task into mandatory, optional and action parts

Important characteristics of a task include:

- The type of the task: either *periodic*, when the task activation pattern repeats itself, or *sporadic*, when the task is triggered once in response to a particular event.
- The *worst-case execution time* of both the mandatory and the action parts of the task.
- The temporal characteristics of the task: *period* and *deadline* for a periodic task, *minimum interarrival time* and *deadline* for a sporadic task.
- The name of the *intention structure* to be used to execute the optional part. This structure is described in [Lalanda et al. 92].

The REAKT Kernel relies on a two-layer control strategy, with a first level in charge of scheduling the real-time mandatory and action parts of each task, and a second level responsible for optimising the solution quality by running optional parts in the remaining time. The remaining sections of this paper focus on the algorithms used in the real-time control layer.

4. THE REAL-TIME CONTROL LAYER

The role of the real-time control layer is twofold: it must guarantee the response time of the mandatory and action parts of all tasks in an application, but should also maximise the amount of time available for optional second-level activities, in charge of improving

the solution quality. For a given task, the basic principle is to first execute the mandatory part, then the optional part, and delay as much as possible the beginning of the action part, while of course executing it before the task deadline.

The scheduling algorithm used in the real-time control layer is based on the *slack time server* algorithm [Lehoczky and Ramos-Thuel 93], as its characteristics are close to the REAKT requirements:

- The slack time server algorithm original goal is to maximise the CPU allocation to soft aperiodic tasks, which is more or less equivalent to maximise the amount of time available for optional processing.
- The basic principle of the algorithm is to delay as much as possible the execution of periodic tasks while guaranteeing their deadlines. This is exactly what needs to be achieved in REAKT for action tasks.
- The algorithm is an optimal one for the deadline-monotonic priority assignment, in the sense that no other algorithms can preserve more time for optional processing.
- The slack time server algorithm is able to recover CPU time unused by stochastic real-time tasks, i.e. tasks which execute faster than their worst-case execution time.

The slack time server algorithm must be adapted to the REAKT task model for two main reasons:

- In the original algorithm, periodic and aperiodic tasks are considered independent. This is not the case in the REAKT task model, as dependencies exist between the mandatory, optional and action parts of a task.
- Existence of sporadic tasks is not taken into account in the original algorithm.

The slack time server approach is based on the creation of a high-priority process, the Slack Server, which is in charge of computing the amount of optional time (slack time) available at a given time. When slack time is available and some optional tasks are ready to execute, the Slack Server pre-empts all real-time tasks to let the lower priority optional tasks run. The Slack Server must precisely know the amount of slack time which is available, in order to resume in time the pre-empted tasks. The problem is then to determine the algorithms which allow the Slack Server to compute the available slack time at each instant.

Three issues must be considered:

- The first one describes the basic principles of the slack time server algorithm when considering only a periodic process set.
- The second part is devoted to the links between the Slack Server and the second layer of scheduling.
- Finally, approaches to scheduling sporadic tasks are considered.

5. THE SLACK TIME SERVER ALGORITHM

1. Principles

Consider a set of N periodic tasks, τ_1, $\tau_2 \ldots \tau_N$. Each task τ_i (C_i, T_i, D_i, O_i) is characterised by its worst-case execution time C_i, its period T_i, its deadline D_i and its initial release time O_i. This model is compatible with the REAKT task model, assuming that:

- only the mandatory and action parts of a task are executed, with an execution time equal to the sum of the mandatory and action part worst-case execution times,

- sporadic tasks behave like periodic ones, with a period equal to their minimal interarrival time (worst-case behaviour).

The tasks are indexed in decreasing priority order, with priorities assigned to tasks according to the deadline-monotonic algorithm [Leung and Whitehead 82], i.e. the task with the shortest deadline receives the highest priority. One additional requirement is that all periods in the system must be integral multiples of some time unit, thus allowing to determine the hyperperiod H of the system (H is the least common multiple of all periods in the system). The hyperperiod is the minimal time after which the task activation pattern of the system repeats itself.

In order to compute the slack time, i.e. the maximum delay that can be inflicted at a given time to the whole task set without missing any deadlines, it is first necessary to compute the maximum delay that can be inflicted to a task τ_i.

At time *t*, the total workload $W_i(t)$ at priority level *i* is given by:

$$W_i(t) = \sum_{j=1}^{i} \left(\left\lceil \frac{\max(0, t - O_j)}{T_j} \right\rceil C_j \right)$$

The workload $W_i(t)$ gives the cumulative work that has arrived for priority 1 to *i* in the time interval [O, *t*].

Now consider the *j*th execution of τ_i in the hyperperiod: the job release time R_{ij} is given by $O_i+(j-1)T_i$ and the job deadline D_{ij} is $R_{ij} + D_i$. A necessary and sufficient schedulability condition (given in [Lehoczky et al. 89]) when using the DM priority assignment is:

$$\min_{t \in [R_{ij}, D_{ij}]} \left\lceil \frac{W_i(t)}{t} \right\rceil \leq 1 \text{ which is equivalent to: } \max_{t \in [R_{ij}, D_{ij}]} \left\lceil t - W_i(t) \right\rceil \geq 0 \qquad (1)$$

The computation of the maximum value for the above function does not need to be done on the continuous interval $[R_{ij}, D_{ij}]$, but only on the set of level-*i* scheduling points for the interval. A level-*i* scheduling point is defined as a time at which one or more tasks of priority higher than *i* are released. The set of scheduling points SP_{ij} for the *j*th job of τ_i is given by:

$$SP_{ij} = \bigcup_{m=1}^{i} \left\{ O_m + k.T_m, k = 0, \ldots \lfloor T_i / T_m \rfloor \right\} \cap \left[R_{ij}, D_{ij} \right]$$

As the workload $W_i(t)$ is an monotonous increasing step function (with jump points given by the scheduling points), the difference $t - W_i(t)$ is a piecewise increasing function, which reaches a local maximum just before a jump point. Therefore, it is correct to search the maximum value only at the scheduling points. In fact, it is also necessary to consider the value of $t - W_i(t)$ at the deadline D_{ij}, if it does not coincide with a scheduling point.

From the above considerations and equation (1), it comes that the maximum delay S_{ij} that can be inflicted to the task τ_i before the completion of its *j*th job is given by:

$$S_{ij} = \max_{t \in SP_{ij}} \left[t - W_i(t) \right] \qquad (2)$$

S_{ij} is the maximum value which can be added to the workload $W_i(t)$ while still verifying equation (1). The series of S_{ij} is non-decreasing for a given *i* if the task set is schedulable.

However, S_{ij} is not the amount of optional processing that can be performed without missing the deadline of the jth execution of τ_i: instead, S_{ij} is the amount of work that can be performed at priority level lower than i. This includes of course optional processing, but also computation time of lower priority tasks and possibly idle time (if no optional work is ready when slack time is available). Therefore, we can introduce the level-i inactivity $I_i(t)$ as:

$$I_i(t) = \sum_{j=i+1}^{N} C_j(t) + O(t) + I(t)$$

where $Cj(t)$ is the time spent in task τ_j, $O(t)$ is the amount of optional processing already performed and $I(t)$ is the time during which the processor has been idle.

The level-i slack time (i.e. the maximum amount of optional processing which can be performed while meeting task τ_i deadlines) is thus given at any time by:

$$S_i(t) = S_{i,\phi_i(t)} - I_i(t) \qquad (3)$$

where $\phi_i(t)$ gives the job number of τ_i at instant t.

The global slack time at instant t $S(t)$ is obtained by taking the minimum value of all level-i slack time at that instant. This is the *maximum* value by which all tasks in the task set can be delayed and still meet their deadlines.

$$S(t) = \min_{i=1\ldots N} S_i(t) \qquad (4)$$

The formulas given above show that the computations required to obtain the slack time are relatively complex. Fortunately, most of these computations can be performed off-line: the series S_{ij}, obtained by applying equation (2), involve only compilation time information, such as periods, deadlines and worst-case execution times. Therefore, a two-dimension table (the Slack Table) can be built off-line with these values, allowing fast access to the information at runtime. One drawback of the Slack Server approach is that it may request a large amount of memory: the size of the Slack Table can become quite large, as the first dimension is the number of jobs of τ_1 in the hyperperiod (assuming τ_1 has the shortest period of all tasks, and therefore the largest number of jobs in the hyperperiod), and the second dimension is the number of tasks in the task set. In REAKT, given the amount of memory requested by some AI components, the size of the Slack Table is not considered a major limitation.

2. Runtime behaviour

At runtime, the basic Slack Server algorithm is as follows:

1. compute the available slack time;
2. if slack time is available, pre-empt all periodic processes to let optional tasks run. After all the available slack time has elapsed, resume all periodic processes;
3. wait until more slack time is available and return to step 1.

The two main points to be discussed in the above algorithm are:

- the slack time computation;
- the time at which more slack time becomes available.

Equation (4) gives a formula to compute the slack time at instant t. In order to transform this formula into an algorithm, it is necessary to introduce a few counters:

- *Idle* will hold the cumulated idle time since the beginning of the hyperperiod;

- *Optional* will hold the cumulated optional processing time since the beginning of the hyperperiod;
- C_i will hold the cumulated time spent in task τ_i since the beginning of the hyperperiod.
- n_i will hold the number of completed executions of task τ_i in the hyperperiod.

All counters are reset to 0 at the beginning of each hyperperiod. Each time the processor finishes an activity (idle, optional or periodic processing), the time spent in the activity is added to the corresponding counter. Each time a task τ_i completes an execution, the counter n_i is incremented.

The algorithm is the following:

```
LET inactivity = Idle + Optional
    min_slack_time = infinity // a large value

FOR i = Ntasks to 1 DO
  slack_time = slack_table[i][ni] - inactivity
  IF slack_time < min_slack_time THEN
    min_slack_time = slack_time
  ENDIF
  inactivity = inactivity + Ci
ENDDO

RETURN min_slack_time
```

This algorithm is a simple minimum value computation. The main point to note here is that the computation of the level-*i* inactivity $I_i(t)$ is performed incrementally by starting the minimum value computation from the lowest priority process, thus saving an iteration. The complexity of this algorithm is O(n).

The second point to study is the time at which more slack time becomes available. Once the currently available slack time is exhausted, more slack time will become available only after the *blocking task* (i.e. the task which was responsible for the minimum slack time value in the previous slack time computation) finishes.

Proof: Let *b* be the priority level responsible for the minimum slack time value S_b in the last slack time computation. After exhaustion of this slack time, all level-*i* slack times S_i are reduced by S_b (after equation (3), since the optional processing time has been increased by S_b). In particular, available slack time at level *b* is 0. Available slack (which is the minimum of all level-*i* slack times) will remain 0 until the level-*b* slack time increases. Equation (3) shows that level-*b* slack time can increase only when S_{bj} increases (as level-*i* inactivity can never decrease), i.e. when the job number of τ_b changes. Therefore, it is necessary to recompute the slack time only after the completion of the current execution of τ_b.

3. Stochastic tasks

As mentioned previously, the slack time server algorithm can be adapted to the recovery of unused CPU time after the completion of a stochastic task.

When stochastic execution occurs, the workload at priority level *i* $W_i(t)$ is overestimated, since it is based on the static worst-case execution time of the tasks. The real workload $W'_i(t)$ is given by:

$$W'_i(t) = \sum_{j=1}^{i} C_j(t) = \sum_{j=1}^{i} \left(\left\lceil \frac{\max(0, t - O_j)}{T_j} \right\rceil WC_j - \Delta C_j(t) \right) = W_i(t) - \sum_{j=1}^{i} \Delta C_j(t)$$

where $C_i(t)$ denotes the real cumulated execution time of task τ_i (including expected computation times of task executions already started but no yet completed), WC_i is the static worst-case execution time of task τ_i and $\Delta C_i(t)$ its cumulated saved time in *completed* executions. It is necessary to consider only completed executions, as there is no way to know how much will be saved in an on-going or future task execution.

The real maximum delay S'_{ij} which can be inflicted to the jth job of task τ_i is given by:

$$S'_{ij} = \max_{t \in SP_{ij}} \left[t - W'_i(t) \right]$$

The above formula uses information ($W'_i(t)$) which can only be known at runtime, but it is clear that computing dynamically the series S'_{ij} is not a practical solution. Therefore, an approximation of the formula is needed.

Let P_{ij} be the first scheduling point in SP_{ij} for which the maximum value S_{ij} is obtained, and let the additional slack time $\Delta_i(t)$ be:

$$\Delta_i(t) = \sum_{j=1}^{i} \Delta C_j(t)$$

$\Delta_i(t)$ is a non-decreasing function, therefore: $\Delta_i(P_{ij}) \geq \Delta_i(t), \forall t \leq P_{ij}$

From the definition of P_{ij}: $S_{ij} = P_{ij} - W_i(P_{ij})$

From the above properties, it can be deduced that:

$$\forall t \leq P_{ij}, S'_{ij} = \max_{t \in SP_{ij}} \left[t - W'_i(t) \right] \geq P_{ij} - W'_i(P_{ij}) = P_{ij} - W_i(P_{ij}) + \Delta_i(P_{ij}) \geq S_{ij} + \Delta_i(t)$$

In order to consider $S_{ij}+\Delta_i(t)$ a valid approximation for S'_{ij} (it will remain a valid approximation as long as it remains below the exact value of S'_{ij}), it is necessary to prove that the formula will be used only for values of t smaller than P_{ij} The formula is used only for the jth execution of task τ_i. Therefore, if it can be shown that the jth job of τ_i completes before P_{ij}, the approximation will be valid.

The proof is by contradiction: let t_1 be the time $\Delta_i(t)$ was last computed ($t_1 < P_{ij}$), and suppose that:

$$\forall t, t_l \leq t \leq P_{ij}, t < W'_i(t) + S_{ij} + \Delta_i(t_l)$$

The above condition states that the total workload (periodic workload plus computed delay) of the system at level i is higher than the available time, and thus that the jth job of τ_i does not complete before P_{ij} . Since $t \geq t_1$, $\Delta_i(t) \geq \Delta_i(t_1)$. The above equations implies that:

$$\forall t, t_l \leq t \leq P_{ij}, t < W'_i(t) + S_{ij} + \Delta_i(t) = W_i(t) + S_{ij}$$

This is in contradiction with the definition of P_{ij}: $P_{ij} = S_{ij} + W_i(P_{ij})$

Equation (3) can now be updated with the above approximation:

$$S_i(t) = S_{i,\phi_i(t)} + \sum_{j=1}^{i} \Delta C_j(t) - I_i(t) \qquad (3')$$

The algorithm used to compute the available slack time must be modified according to the above formulas. It is first useful to define a new set of counters ΔC_i which will hold the cumulated saved time since the beginning of the hyperperiod. The counters are reset

at the beginning of each hyperperiod and the relevant counter is updated at the end of each task execution.

The algorithm for computing the slack time is now:

```
LET inactivity = Idle + Optional, min_slack_time = infinity

FOR i = Ntasks to 1 DO
  slack_time = slack_table[i][ni] - inactivity
  IF slack_time < min_slack_time THEN
    min_slack_time = slack_time
  ENDIF
  min_slack_time = min_slack_time + ΔCi
  inactivity = inactivity + Ci
ENDDO

RETURN min_slack_time
```

As in the previous algorithm, it is important to note that the computation of the additional slack time $\Delta_i(t)$ is done incrementally and does not require an additional iteration. The complexity of the algorithm remains O(n).

The property that additional slack time becomes available only when the *blocking* task completes is not true any more in case of stochastic executions: it is clear in this case that the early completion of a non-blocking task allows the system to recover some additional slack time. Therefore, the slack time computation algorithm must be executed after the completion of each process finishing early.

6. LINKS WITH THE SECOND SCHEDULING LAYER

1. Principles of interactions

In order to understand the interaction between the Slack Server and the REAKT components used in the second scheduling layer, it is useful to first list all relevant processes present in the REAKT Kernel, following decreasing priorities:

- The process with the highest priority is the **Slack Server**.
- Periodic and sporadic **real-time tasks τ_i** , $i = 1 \ldots N$, sorted by decreasing priorities.
- The **Intention Scheduler** schedules the optional tasks (intentions) according to their deadline and importance. The complete scheduling algorithm used by the Intention Scheduler is beyond the scope of this paper. It is based on the earliest-deadline-first algorithm, and takes into account extensions proposed by [Chetto et al. 90].
- Optional parts of tasks are implemented as **intentions**.

All the activities which execute at a priority level below those of the real-time tasks can only run while these real-time tasks are suspended by the Slack Server. The basic system cycle is the following:

- When slack is available, the Slack Server suspends itself and all real-time tasks;
- The Intention Scheduler is activated and can be in the following states:
  - *No intentions are ready to execute* (all intentions are pending): Control is immediately returned to the Slack Server, which will resume the real-time tasks. This allows to preserve the slack time for later optional processing. The Slack Server needs to be activated when new information (e.g. events) able to trigger optional processing is sent to the Intention Scheduler, in order to compute the currently available slack time and let the optional processes run.
  - *Some new intentions are ready* and need to be inserted in the schedule: This insertion is performed following the second-layer scheduling algorithm, using a function returning the amount of slack time available between *now* and the

deadline of the intention. The specification of this function is given below. Once intentions have been scheduled, the algorithm can proceed to the following step.

- ♦ *The schedule is not empty but the first intention is not active*: This happens when the mandatory part associated to the intention has not been executed yet. In this case, the Intention Scheduler requests the Slack Server to resume only the specified mandatory task. Explicitly resuming real-time work from the second scheduling layer is the way of implementing at runtime the precedence constraints of the REAKT task model.
- ♦ *The first intention in the schedule is active*: it is executed while slack time is available.

- When the slack time is exhausted, the Slack Server resumes all real-time tasks, thus pre-empting lower priority activities.

The interactions between the Slack Server and other REAKT components (mainly the Intention Scheduler) require two types of modifications in the slack time server algorithms:

- The basic loop of the Slack Server process must be modified, in order to react to messages sent by the Intention Scheduler.
- The slack time computation algorithms must be extended to take into account the possibility of executing real-time tasks while slack time is available: this can occur when the Intention Scheduler requests the execution of the mandatory part of a ready intention.

2. Modified Slack Server loop

The Intention Scheduler can send three types of messages to the Slack Server:

- A notification that no more optional work is ready. The Slack Server should then resume all real-time tasks, in order to preserve available slack time for later use. This message can only be received while slack time is available, i.e. while the Slack Server is sleeping, waiting for the time-out to arrive.
- A notification that new optional work is ready. This is the opposite message, and the Slack Server should then immediately recompute the available slack time and pre-empt the real-time tasks to start the optional work. This message can only be received while the Slack Server is not sleeping. Messages of this type should be generated only when the Intention Scheduler is idle and either an event is received or a pending intention becomes ready.
- A request for the execution of the mandatory part of a given task. The Slack Server should then recompute the amount of slack time available, taking into account the early execution of the real-time task, and suspend itself for the newly computed duration. This message can only be received while slack time is available.

The above classification shows that steps 2 and 3 of the algorithm presented in section 5.2 should be modified. The modified algorithm is:

1. compute the available slack time;
2. if slack time is available, pre-empt all periodic processes to let optional tasks run. After all the available slack time has elapsed or a message is received from the Intention Scheduler, resume all periodic processes;
3. wait until either more slack time or optional work becomes available and return to step 1.

3. Early execution of mandatory parts

When the Slack Server receives a message from the Intention Scheduler requesting the execution of the mandatory part of a task, it must:

- resume the given task;
- compute the new available slack time, taking into account that the resumed task will execute first. Intuitively, the available slack time (i.e. the time available for executing both the resumed task and optional work) should increase, as the operation is basically an CPU exchange between optional processing and the real-time task. However, the exact amount of additional slack time depends on the priority of the resumed task.

Let r be the priority level of the resumed task. From a scheduling standpoint, executing τ_r explicitly while slack time is available is equivalent to considering that the *current* execution of the task is optional processing and that the computation time of the mandatory part is null (this assumption remains valid while slack time is available). In turn, this implies that the workload for priority levels lower than or equal to r has been overestimated. The real workload $W'_i(t)$ is given by:

$$\forall i \geq r, W'_i(t) = W_i(t) - C_r$$
$$\forall i < r, W'_i(t) = W_i(t),$$

Therefore, the real level-i slack time $S'_i(t)$ is given by:

$$\forall i \geq r, S'_i(t) = S_i(t) + C_r$$
$$\forall i < r, S'_i(t) = S_i(t),$$

The above formula can be straightforwardly derived from the usual slack time formulas (equations (2) and (3)), as the additional term in the equation is constant (it has no effect on the maximum value computation involved in the computation of the series S_{ij}).

Therefore, the loop to compute the available slack time when a real-time task is explicitly resumed is simply divided in two parts: the first part computes the level-i slack for priority levels lower than or equal to r, adding C_r to the slack time, the second for priority levels higher than r.

This algorithm has several interesting properties:

- When the resumed task has a priority lower than that of the blocking process, it is easy to show that no additional slack is available: the minimum value for the slack time, which is given by the level-b slack time (where b is the priority level of the blocking process), remains unchanged, as $r > b$.
- When the resumed task has the highest priority, the additional slack time is equal to the task computation time: since all level-i slack time are incremented by the task computation time, it is clear that the minimum value is also incremented by this amount.
- The property above can be usefully applied to real-time data acquisition tasks. These tasks usually have the highest priorities in the system. In addition, it is clear that they should not be suspended by the Slack Server, since all other tasks (including optional ones) use the data values they produce. Therefore, when a data acquisition task becomes ready while slack time is available, it should be immediately resumed. The above property can then be applied to compute the new available slack time, which should be the last computed slack time incremented by the data acquisition task computation time.

4. Computation of the available slack time in a given time interval

When scheduling optional work, the Intention Scheduler must take into account the computational requirements of real-time tasks: the available time to schedule an intention is equal to the amount of optional processing which can be performed between the intention release time (usually the current time) and the intention deadline.

The approach to compute this value is obviously related to the slack time computation: indeed, the available optional time O is given by:

$$O = \min_{i=1\ldots N} S_i(D) = \min_{i=1\ldots N}(S_{i,\phi_i(D)} - I_i(D))$$

where D is the intention deadline.

However, it is difficult to apply the above formula directly: while it relatively easy to obtain the value of $I_i(t)$ at a given instant, by appropriately updating counters, the computation of this value in the future (which is the case for D) requires to know the number of times each task will complete and be released during the interval [t, D]. Computing dynamically this value would be too costly.

The solution used in REAKT is the following:

- compute the function giving the available slack at any time, assuming that optional processing is always ready. The values of this step function can be computed off-line and stored in an array for each time point in the hyperperiod.
- the available slack time is then given by adding the currently available slack time to the difference between the value of the function at D and the current value of the function. By keeping appropriate time counters, it is possible to compute the available slack time until D without performing any iterations.

One aspect which is not considered in the current version, but on which more work is needed, is the existence of periodic intentions (optional parts of periodic tasks). Clearly, when scheduling an intention, the system should be aware of the workload that will be created by the forthcoming releases of periodic intentions. An off-line computation of a static periodic intention schedule, to be updated dynamically, could be an interesting approach.

7. SCHEDULING SPORADIC TASKS

The general problem of scheduling jointly periodic, sporadic and optional tasks is a difficult one, for which no general solution has been given in the literature: the underlying problem is to determine exactly the maximum amount of optional processing that can be performed, while the exact pattern of sporadic task arrival is by definition unknown. Therefore, there is either the risk to perform too much optional processing, and thus miss hard deadlines if many sporadic tasks are released, or to be too conservative and never perform any optional processing.

Two approaches have been studied for REAKT:

- The first one consists in applying a period transformation to the sporadic tasks, in order to transform them in stochastic periodic ones
- The second one consists in computing the Slack Table assuming that the sporadic tasks behave like periodic ones, and then recover unused CPU time at runtime.

1. Period transformation

This approach consists in transforming a sporadic task τ_i with deadline D_i and computation time C_i into an equivalent task with deadline 1 and computation time C_i/D_i. In this case, the task can be considered as a periodic task which is not always released, and the technique described for stochastic periodic tasks applies.

However, this approach has several drawbacks:

- It generates an overhead, as each task must be controlled by a timer in order to execute exactly for the planned amount of time.
- It may be difficult to apply when tasks use critical sections, as it is usually a good practice to avoid suspending tasks inside a critical section.
- It increases the load of the system, as the tasks are executed at a higher priority than their original one (as their new deadline is smaller than before).

For all these reasons, it has been decided not to implement this solution.

2. Slack time recovery

In the second approach, the Slack Table is computed assuming the worst-case behaviour of the system, i.e. all sporadic tasks are considered periodic. The problem is therefore to recover slack time when the sporadic tasks are not executed. It is different from the problem of recovering unused time in stochastic task executions, as the sporadic tasks are not released are predefined instants, but rather can start at any time, as long as they respect their minimum interarrival time.

A sporadic task σ_i of priority level i affects the value of $S_j(t), j \geq i$, in two ways:

- the computation of the workload $W_j(t)$,
- the computation of the set of scheduling points SP_{jk}.

as both computations depend on the number of releases and the release times of σ_i.

In order to avoid the complete slack time computation at runtime, it is necessary to find an approximation of the slack time which can make use of the values stored in the Slack Table. This in turn requires that the set of scheduling points used to compute each value in the table remains unchanged, as any modification would induce a change in the maximum value computation in equation (2).

The sets $SP_{jk}, j \geq i$, remain unchanged when the release time of σ_i is translated by T_i, the minimal interarrival time for the task. Therefore, the approximation is to consider that slack time can be recovered as soon as the sporadic task has been delayed by T_i, the amount of recoverable slack time being equal to the worst-case computation time C_i. The algorithm defined for dealing with stochastic task execution can then be used to compute the modified slack time.

This approach has been implemented, and experiments have shown that the approximation is very pessimistic, especially in the case of sporadic tasks having large interarrival times. For applications where time constraints are not too strict, dynamic incremental computation of the Slack Table could provide a better solution.

8. CONCLUSIONS

We have presented in this paper some elements of the REAKT Kernel, a software architecture well-adapted to the requirements of real-time knowledge-based applications. The paper focused on the scheduling techniques which allow the combination of predictable and unbounded computation while guaranteeing response

times. The approach is based on the progressive reasoning paradigm, which allows a system to obtain a first solution quickly, and then refine it while time is available.

Activities in the REAKT Kernel are represented with the REAKT task model, which decomposes each activity into three parts, mandatory, optional and action. This model supports the easy integration of reflex behaviour and refinement activities, which is essential in the progressive reasoning approach. A scheduling algorithm based on the slack time server approach allows the system to maximise available time for optional processing while guaranteeing response times of the real-time tasks. This algorithm provides an answer to the complex problem of jointly scheduling periodic, sporadic and optional tasks in a real-time system.

A second scheduling layer, not described in this paper, is used to control the execution of optional processing. This layer can use both complex scheduling techniques, based on deadlines, importance and solution quality, and complex unbounded problem-solving algorithms. Interactions between the two layers is supported, and allows the second scheduling layer to modify, within certain limits, the behaviour of the real-time layer when optional processing requires it.

Validation of the REAKT Kernel has been performed through the development of several applications, including a large alarm management system in an oil raffinery. The development of this particular application has already shown that the REAKT functionalities are well-adapted to the requirements of a typical RTKBS such as a process control system.

9. REFERENCES

[Audsley et al. 91]
Audsley, N.C, Burns, A. and Welling, A.J. " Incorporating unbounded algorithms into predictable real-time systems ", *Technical Report*, Universisty of York, UK, 1991.

[Charpillet and Boyer 94]
Charpillet, F. and Boyer, A. " Incorporating AI Techniques into Predictable Real-Time Systems: REAKT Outcome ", *Proc. 14th International Avignon Conference*, vol. 1, pp. 121-134, Paris, 1994.

[Chetto et al. 90]
Chetto, H., Silly, M. and Bouchentouf, T. "Dynamic Scheduling of Real-Time Tasks under Precedence Constraints." *The International Journal of Time-Critical Computing Systems*., 1990, vol. 2, nr 3, pp.181-194.

[Engelmore and Morgan 88]
Engelmore, R. and Morgan, T. (eds.) *Blackboard Systems* (Addison-Wesley Publishing Company, 1988).

[Fjellheim et al. 94]
R. Fjellheim, T. Pettersen and B. Christoffersen. " A methodology for real-time knowledge-based applications ", *Proc. of the Second World Congress on Expert Systems*, Lisbon-Estoril, January 94.

[Galan et al. 93]
Galán J.J., González-Quel A., Mensch A., Monai F., Kersual D. " An Implementation Analysis of the REAKT Toolkit ", *Workshop on Integration Technology for Real-Time Intelligent Control Systems*, Madrid, Spain, 1993.

[Garvey and Lesser 94]
Garvey, A. and Lesser, V. " A Survey of Research in Deliberative Real-Time Artificial Intelligence "**,** *Real-Time Systems*, 1994, vol. 6, n°3, pp. 317-347.

[Jagannathan et al. 89]
Jagannathan, V., Dohiawala, R. and Baum, L. S. (eds.) *Blackboard Architectures and Applications* (Academic Press, 1989).

[Laffey et al. 88]
Laffey, T.J., Cox, P.A., Schmidt, J.L., Kao, S.M. and Read, J.Y. " Real-time knowledge-based systems ", *AI Magazine*, Spring 88, pp 27 - 45.

[Lalanda et al. 92]
Lalanda, P., Charpillet, F. and Haton, J.-P. " A Real Time Blackboard Based Architecture ", *Proc. 10th ECAI*, Vienna, Austria, 1992.

[Lehoczky et al. 89]
Lehoczky, J., Sha, L. and Ding, Y. " The rate-monotonic scheduling algorithm: exact caracterization and average case behavior ", *IEEE Symposium on Real-Time Systems,* 1989.

[Lehoczky and Ramos-Thuel 93]
Lehoczky J. and Ramos-Thuel, S. " An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems ", *IEEE Symposium on Real-Time Systems,* 1993.

[Leung and Whitehead 82]
Leung, J.Y.-T. and Whitehead, J. " On the complexity of fixed-priority scheduling of periodic real-time tasks ", *Performance Evaluation*, 1982, Vol. 2, pp. 237-250.

[Mensch et al. 94]
A. Mensch, D. Kersual, A. Crespo, F. Charpillet and E. Pessi. " REAKT: real-time architecture for time-critical knowledge-based systems ", *Intelligent Systems Engineering Journal*, Autumn 94, pp 153-167.